The Derivation of Compositional Programs

K. Mani Chandy and Carl Kesselman

Computer Science Department
California Institute of Technology

Caltech-CS-TR-92-18

The Derivation of Compositional Programs

K. Mani Chandy and Carl Kesselman *
California Institute of Technology
Pasadena, California 91125, USA
mani@vlsi.caltech.edu, carl@vlsi.caltech.edu

July 24, 1992

To appear in the Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming, MIT Press.

Abstract

This paper proposes a parallel programming notation and a method of reasoning about programs with the following characteristics:
1. Parallel Composition The notation provides different forms of interfaces between processes; the more restrictive the interface, the simpler the proofs of process composition. A flexible interface is that of cooperating processes with a shared address space; proofs of programs that use this interface are based on non-interference [OG76] and temporal logic [Pnu81, CM88, Lam91]. We also propose more restrictive interfaces and specifications that allow us to use the following *specification-conjunction* rule: the strongest specification of a parallel composition of processes is the conjunction of the strongest specifications of its components. This rule is helpful in deriving parallel programs.
2. Determinism A process that does not use certain primitives of the notation is guaranteed to be deterministic. Programmers who wish to prove that their programs are deterministic are relieved of this proof obligation if they restrict their programs to a certain subset of the primitives.

1 Parallel Composition

VLSI is an example of parallel programming as Chuck Seitz asserts. The success of VLSI is due in part to a hierarchy of interfaces for composing circuits: from composing transistors to form memory units, to composing microprocessors to form multicomputers. A designer putting transistors together has to be concerned with issues such as parasitic capacitance. A designer putting microprocessors together works with a more restrictive interface that does not deal with such details; the interface is such that a microprocessor behaves as a microprocessor regardless of the circuits to which it is connected. Composing transistors and composing microprocessors are both instances of parallel composition; the interfaces between transistors is different from the interfaces between microprocessors, and therefore, the ways of reasoning about compositions of transistors is different from the ways of reasoning about compositions of microprocessors.

What is the analogy to composing processes? There are situations where we may want to employ flexible interfaces between concurrent processes and other situations in which we want to use more restrictive interfaces between processes. In general, more restrictive interfaces allow for simpler proof techniques and more flexible interfaces can provide more efficiency. We propose a notation and methods of reasoning about programs that allow programmers to design different kinds of interfaces between concurrent processes. Programmers can balance flexibility on the one hand with ease of reasoning on the other, in designing their own interfaces.

A very flexible interface between concurrent processes is where processes share variables, and use atomicity and **await** commands [OG76]; in this case methods of reasoning are based on non-interference [OG76] or temporal logic and its derivatives [Pnu81, CM88, Lam91]. More restrictive interfaces and specifications allow us to use the powerful specification-conjunction rule for reasoning about parallel composition.

Consider a simple example that illustrates the problem of interfaces between concurrent processes.

Example The process p:

```
do x = 0 → x := x+2   []   x = 1 → x := -1 od
```

satisfies the specification: (x = 0) $\leadsto$ (x $\geq$ 2), if x = 0 at any point in its computation, then at a later point in its computation x $\geq$ 2.

The same specification is satisfied by the process q:

```
do ((x = 0) ∨ (x = 1))  → x := x + 1 od
```

The parallel composition of p and q, p||q, does not satisfy the specification, because q can change the value of x from 0 to 1, and then p can change the value of x to -1, after which x remains unchanged.

The parallel composition of processes, all of which have a common specification R, may not satisfy specification R, because the shared-memory interface between processes is flexible and allows one process to "interfere" with the proof of another [OG76]. Later, we define more restrictive interfaces and specifications that permit simpler proofs.

1.1 Processes

We define processes in terms of transition systems [Pnu81, Lam91, CM88]. A system is a set $\mathcal{P}$ of processes and a set $\mathcal{G}$ of global variables. A process is (i) a set L of *local variables*, (ii) a set G of *shared variables* where $G \subseteq \mathcal{G}$, (iii) the *initial values* of its variables $L \cup G$, and (iv) a set of atomic *actions*. The name space of local variables of a process is local to the process; by contrast, the name space of shared variables is global to the system.

A state of a process is defined by the values of the variables of the process. (Program counters or other methods of representing the locus of control are treated as variables, as in [CM88].) The initial state of a process is given by the initial values of its variables.

An action is a binary relation on states of the process. We shall say that an action A takes a process from a state S to a state S' if and only if $(S, S') \in A$. An action A is *executable* in a state S if there exists an S' such that A takes the process from S to S',

1.2 System States and Transitions

The state of a system is a tuple of states of its component processes where the values of shared variables are consistent among all processes, i.e., if v is shared by p and q, and $v = v'$ in the state of process p in the tuple, then $v = v'$ in the state of process q in the tuple as well.

The initial state of a system is a tuple of initial states of its component processes if the values of shared variables are consistent in the tuple. If initial shared-variable values are inconsistent, the initial system state is undefined.

Transitions between system states are labeled with actions of component processes. There exists a transition labeled A from a system state S to a system state S' if and only if there exists a transition A in a component process p such that

1. values of all variables other than those referenced by p are identical in S and S', and
2. action A takes the values of variables referenced by p from their values in S to their values in S'.

1.3 Computations

A *computation* of a process p is an initial state S_0 of the process, and a sequence of pairs (A_i, S_i) where $i > 0$ and action A_i takes the process from process-state S_{i-1} to process-state S_i, and the sequence satisfies the following *fairness rule*:

For all infinite computations, if action B is executable at some point in the computation then there is a later point in the computation at which either B is executed or B is not executable.

B is executable in $S_i \Rightarrow$
$(\exists j : j > i : (A_j = B) \vee (B \text{ is not executable in } S_j))$

A *terminal* process-state is a state S such that all actions of the process are disabled in S. A *maximal computation* of a process is either an infinite computation or a computation that ends in a terminal state.

System computations, terminal system states, and maximal computations of systems are defined in the same way as for processes (except that system states replace process states in the definitions).

1.4 Process Properties and Open Systems

A conventional definition of process *properties* is as follows:

Closed-System Definition of Properties
A property R of a process p is a predicate on maximal computations of p where all maximal computations of p satisfy R.

With this definition, $p||q$ does not have a property common to both p and q. How can we define process properties so as to use the following rule:

A property of p is a property of $p||q$, for all processes q?

An obvious solution is to redefine properties in a somewhat unconventional way:

Open-System Definition of Properties
R is a property of p if and only if for all processes q, R is a predicate on maximal computations of $p||q$, and R holds for all maximal computations of $p||q$.

The conventional definition of process properties is sometimes referred to as the *closed-system* definition, and the alternative definition is called an *open-system* definition; this nomenclature is because the conventional definition defines properties of a process executing in isolation, whereas the alternative definition defines properties of a parallel composition of a process with some arbitrary "environment."

Relative Advantages of Open and Closed Systems The primary disadvantage of the open-system definition is that the properties that we can prove about open-systems are weak. To prove a property of a process p we have to consider computations of p executing concurrently with q, for *all* processes q. So, we are forced to consider processes that the designer of p had no intention of composing with p. For instance, we cannot prove that a multiplier circuit multiplies because it can be connected to a megavolt power supply that fries the multiplier!

The primary disadvantage of the closed-system definition is that we do not enjoy the benefits of specification-conjunction. When we wish to prove properties about the parallel composition of processes we use noninterference [OG76] or prove properties from the *text* of the component programs [CM88] as opposed to the preferred mode of composing specifications without regard to program text.

2 The Proper Interface Approach

An approach that enjoys some of the advantages of both open and closed systems is the *proper interface* approach. We define a *proper* interface (or protocol) by which processes cooperate. We restrict attention to process composition in which the interface between processes is proper; we call composition with proper interfaces *proper composition.* We define process properties as for open systems, but we restrict attention to proper interfaces:

Proper-Interface Definition of Properties
A property of a process p is a predicate on maximal computations of $p||q$, that holds for all maximal computations of $p||q$, for all q *such that the interface between* p *and* q *is proper.*

Are the properties we can prove, using proper interfaces, too weak to be useful? That depends on the definition of proper interfaces — the more flexible the interface, the weaker the properties.

One of the advantages of hardware modules is that engineers have developed a set of proper interfaces. A hardware module is specified in terms of its inputs and outputs for a proper interface. When hardware modules are composed, the designer proves that the interfaces are proper (and this is usually straightforward) and then the designer can use specification-conjunction. Design is simplified greatly by being able to assert that the output of a multiplier circuit is the product of its inputs, regardless of the circuits with which the multiplier is composed, provided that the interfaces are proper. The designer of a multiplier circuit does not have to be concerned about the circuit being connected to a megavolt power supply because such an interface is not proper. The designer has to be concerned, however, with *all* possible environments with proper interfaces.

A problem with concurrent programming is that we do not usually specify software processes in terms of standard interfaces with clearly defined inputs and outputs; and we define process properties in terms of closed systems; and, therefore, we cannot use specification-conjunction to prove properties of concurrent programs. A proper-interface approach is particularly helpful in designing libraries of processes, all of which use the same interface.

2.1 A Collection of Proper Interfaces

For an open-systems specification, we specify an interface of a process in terms of the outputs of the process and the outputs of the environment of the process. The form of outputs (messages, shared-variables,...) is not important at this stage. There are many ways of designing interfaces, but to simplify design we will design processes and proper interfaces that satisfy the following rules.

Rule 1: An action is one of the following three types:

1. **Inputs:** The action reads shared variables as input and (possibly) reads or modifies local variables.

2. **Outputs:** The action modifies shared variables as output and (possibly) reads or modifies local variables.

3. **Internal:** The action does not reference shared variables.

The output actions and internal actions of a process are nonblocking because they depend only on the state of the process (and are otherwise independent of the state of the system).

Rule 2: If an input action B is executable at some point in a computation, then it remains executable until it is executed.

$(B \text{ is executable in } S_i) \wedge (B \neq A_{i+1}) \Rightarrow (B \text{ is executable in } S_{i+1})$

This rule disallows probes [Mar85] and other nonmonotonic operators on inputs. A probe checks whether an input is present and takes some action if there is no input; this action can be disabled when an input arrives. But, according to the rule, if an action is executable, it must remain executable until the action is taken.

This rule also prohibits a process from changing an earlier output value; a process can *add* to its earlier output but it cannot change its earlier output. Thus, we have an ordering relation on the "length" of outputs and inputs. For now, assume that outputs and inputs are sequences of values. We can consider other data structures such as trees, provided "length" is defined properly, but this is not central to our discussion.

Rule 3: An input of a process is a prefix of an output of at most one process.

If an input to a process were an output of two or more processes, we would have to deal with interference between processes writing to the same input.

The input to a process may not equal the output from a process because of delays in transmission; hence, we require the input to be an initial subsequence of the output.

An output can feed an arbitrary number of inputs. If x is a process output, and y and z are process inputs, we can have:
(x is a prefix of y) $\wedge$ (x is a prefix of z)

Consider the example, given earlier, of processes p and q sharing a variable x, where though both p and q have a property R, the parallel composition $p||q$ does not have property R. What are the inputs to p? One definition is that the inputs to p are the sequence of values of x prior to actions by p; these are the sequence of values of the shared variable, projected on p's computation. A definition of the outputs of q are the values of x at the termination of actions of q. But, with this definition, the rules for inputs and outputs are not satisfied! One process can modify x with no impact on the computation of the other process. There seems to be no convenient way to define inputs and outputs so that the input of one process is a prefix of the output of the other.

Next we propose a few proper interfaces that satisfy the rules.

2.2 Examples

Modify Privileges At most one process has the privilege of modifying a shared variable. The modify-privilege for a shared-variable can be passed between processes; the methods by which privileges are passed is not important at this point in the discussion. An input of a process p, and an output of a process q is the sequence of values of a shared variable at the points in the computation at which the modify-privilege for the shared variable is passed from q to p.

Single-Assignment Associated with each shared variable `x` is a boolean `x.assigned` which is initially `false`. When a value is assigned to `x`, the boolean `x.assigned` becomes `true` — i.e., a postcondition to every assignment to `x` is `x.assigned`.

A value can be assigned to a shared variable at most once in a computation; therefore, if the precondition to an assignment to `x` is `x.assigned` holds, then the postcondition is that `error` holds, where `error` is a boolean that indicates whether an error has occurred.

The booleans `x.assigned` cannot appear in the program text. Note that rule 2 prohibits testing whether a variable is unassigned.

The execution of a process reading an unassigned shared variable is suspended until the variable is assigned a value. Each shared variable referenced by a process is either an input or an output variable of the process, and a shared variable is an output variable of at most one process.

An output (input) of a process is the value (if any) assigned to an output (input) variable of the process

Computations of unbounded length are achieved by using data structures, such as lists, of unbounded length.

Message Passing The shared variables are first-in-first-out channels. The state of a channel is a queue of messages. The length of the queue is unbounded. A channel is empty initially. At most one process can send messages on a channel (append to the queue) and at most one process can receive messages on a channel (delete from the queue). Sending is nonblocking — i.e. the executability of a send action of a process p depends only on the state of p. Receives are blocking — a receive on a channel is executable only if the channel is nonempty. Probes are not permitted: a channel cannot be tested to determine if it is empty.

An output of a process p is the sequence of messages that p sends on a channel. An input of a process p is the sequence of messages received by p on a channel.

The privilege to send messages, and to receive messages, on a channel can also be sent from one process to another [FC92].

2.3 Reasoning about Programs

A property of a process is a temporal logic formula, and the only rule we have for parallel composition is: if R is a property of p then R is a property of $p||q$, for any q such that the interface between p and q is proper.

Because, we have an ordering on the lengths of inputs and outputs, an operator that is useful is *establishes* [CT91]. Let R be a predicate on process

states. Process p *establishes* R if and only if for all maximal computations of $p||q$, where q is any process such that the interface between p and q is proper, there exists a suffix of the computation such that R holds for each state of the suffix.

In temporal-logic terms, p establishes R means "eventually always R."

The proof that establishes is conjunctive is straightforward [CT91].

(p establishes R) $\wedge$ (p establishes T) $\Rightarrow$ (p establishes $R \wedge T$)
(p establishes R) $\wedge$ (q establishes T) $\Rightarrow$ ($p||q$ establishes $R \wedge T$)

The following example illustrates the use of *establishes*.

Consider a single-assignment interface. Process `p` has inputs `x` and output `y`. Process `q` has inputs `x` and `y` and output `z`. The body of `p` is: `y = x+1` and the body of `q` is `z = x*y`

We can prove:

`p` establishes (`x.assigned` $\Rightarrow$ `y.assigned` $\wedge$ `y = x+1`)
`q` establishes ((`x.assigned` $\wedge$ `y.assigned`) $\Rightarrow$ `z.assigned` $\wedge$ `z = x*y`)

Using specification-conjunction and predicate calculus:

$p||q$ establishes
((`x.assigned` $\Rightarrow$ `y.assigned` $\wedge$ `z.assigned` $\wedge$ `y = x+1` $\wedge$ `z = x*y`)

The use of *establishes* simplifies proofs of parallel composition with proper interfaces. The operator *establishes* was proposed within the context of the PCN theory. Here, we observe that the same constructs can be extended to other proper interfaces.

3 Determinism

We can prove that if each process in a parallel composition is deterministic, and the parallel composition satisfies our 3 rules for proper interfaces, then the parallel composition is deterministic as well: Different executions of the parallel composition produce identical output.

4 Programming Languages and Proper Composition

Next, we consider language support for the design of families of interfaces for parallel composition. We wish to support flexible interfaces with which we use closed-systems specifications and we also wish to support more restrictive interfaces with which we use proper-interface specifications.

We have based our research on the C++ programming language [ES90]. A major objective of C++ is to provide a language framework for constructing program libraries with well defined, compiler enforced interfaces. These features, along with its widespread, use motivated our choice of C++. Our design methodology is supported by C++ augmented by small number of simple extensions. We call the resulting language Compositional C++ or CC++. A detailed discussion of CC++ can be found in [CK92].

Parallel composition in CC++ is provided by parallel blocks (equivalent to parbegin/parend) and a parallel loop construct. Any statement can appear in a parallel block; blocks can be nested. The execution of a parallel block terminates when all statements in the block terminate.

A generalization of the single assignment rule is used to synchronize operations between statements executing in parallel. Primitive data types can be declared to be synchronization or **sync** objects. A process reading an uninitialized **sync** object suspends until the object is initialized by an assignment. Multiple initialization of the same variable is an error. CC++ generalizes single assignment variables in that *user-defined* data types can also be made **sync**. The designer of the data type has complete control over the semantics of user defined **sync** objects and the operations that can be performed on such a data object.

In C++, one can associate a function with a user-defined data type; such a function can only be applied to an object of the appropriate type. These functions control the manner in which a data type can be used. Such functions are commonly invoked through a pointer to an object of that data type. If a pointer to an object is a global variable of a system, invoking a function through such a pointer corresponds to a remote procedure call. If a reference to an object is shared by more than one statement in a **par** block, nondeterministic execution can result. As part of the interface specification for a data type, we can indicate that the operations of a function take place

atomically.

5 The Relationship between CC++ and Logic Programming

In our work, we have focused on language mechanisms that facilitate the design of interfaces for parallel composition. The design of CC++ draws ideas from a wide range of parallel programming languages. These include data flow languages with single-assignment variables [TE68, Ack82], remote procedure calls [TA90], message passing [Sei91], actors [Agh86], concurrent logic programming [FT90, Ued86, Sha86] and compositional languages, particularly PCN [CT91]. While a range of comparisons are possible, the following discussion will focus on the relationship between CC++ and concurrent logic programming languages.

A "pure" logic program has a declarative reading. Such a program does not presuppose any ordering on the actions the program performs. The execution of a program produces a consistent set of variable bindings. As long as the bindings are consistent between program components, the order in which the bindings are determined is not specified. Thus, the conjunction and disjunction operators in a logic program can be viewed as specifying a parallel composition. Clearly, in a pure logic program, one that does not utilize predicates with side effects, all compositions are proper. One may write an open-system specification for a program component, however, that specification is restricted to use only logical variables. If predicates with side effects are used (i.e. such as cut, input/output, assert), then the specification must be weakened.

The situation in the committed choice languages such as Strand [FT90], FCP [Sha86], GHC [Ued86] or Parlog [Gre87] is not as clear cut. In these languages, only one solution path is explored, there is no backtracking or or-parallel search. In order to control which solution path is followed, modify access to variables is restricted. A consequence of read only variables is that the programmer has additional proof obligations, or the open-system specification is weakened. For example, a procedure can deadlock if the environment with which it is composed does not follow an appropriate resource acquisition protocol.

Concurrent logic programming languages provide the safety net that all programs written in such a language conform to the protocol of a proper declarative composition. By contrast, CC++ places the burden of designing interfaces and their proofs on the programmer. We observe, however, that in many large scale parallel programs, efficiency and system concerns dictate that some parts of the program be written in an imperative programming language. Indeed multilingual programming using a concurrent logic programming language as the interface had been proposed as a useful parallel programming paradigm [FO90, FO91, FT90, CT91] and most logic programming languages include "foreign language" interfaces. However, once foreign language components are introduced into a system, the tasks of designing interfaces and their proofs falls back onto the user.

It is important to recognize that a `sync` variable in CC++ is a pure single assignment variable and not a logical variable. In particular, the assignment `x = y` suspends until y has a value; variable-to-variable assignments are not made. Consequently, structured `sync` data behaves more like an I-Structure [AT80] from the dataflow language Id [Ack82] than a tuple from a logic programming language. The use of single assignment variables in place of logical variables has the advantages that assignment semantics are completely consistent with C++, and that pointer dereferencing is not required prior to variable use. The disadvantage is that some concurrent logic programming techniques, such as the short circuit technique [Tak89] become sequentialized. This is not a significant drawback, however, because termination of parallel blocks is easily determined.

CC++ has many ideas in common with the parallel programming language PCN [CT91] which in turn draws heavily from committed choice concurrent logic programming languages such as Strand [FT90]. There are, however, fundamental differences between them. These include:

- CC++ provides a general shared memory model. This includes having pointers to data objects.
- PCN permits `x = y` as an equality. CC++ treats all assignment operators as assignment of value.
- Remote procedure call is a primitive operation in CC++.

- There are no nondeterministic language constructs in CC++ as opposed to PCN. Nondeterminism in CC++ is obtained through interleaving of atomic actions.
- The emphasis in CC++ is on the development of families of interfaces and proofs. PCN provides a single-assignment interface and proof theory.

6 A Programming Example

To demonstrate how CC++ supports parallel program design through proper interfaces, we present a simple example. The parallel program we wish to construct is a producer/consumer system. The `producer` process produces a sequence of values. The values are processed in order by a `consumer` process. Both the producer and the consumer execute in parallel. One of the advantages of CC++ is that the parallel code is quite similar to the sequential C++ code that solve the same problem. The primary difference is the use of `sync` variables and the introduction of parallel blocks.

We will solve this problem using three different interfaces: i) a declarative interface, ii) a modify-privileges interface, and iii) a message passing interface.

Figure 2 shows how a producer/consumer program is constructed using a declarative interface. The sequence of values is passed from the producer to the consumer on a list. The list structure, whose declaration is shown in Figure 1, is declared so that both the value being placed on the list, and the pointer to the next cell of the list are `sync`. The producer iterates, creating new list cells, initializing their values and setting the next field of the previous cell to point to the newly created cell. The consumer is passed a `sync` pointer to a list cell. It cannot proceed until that pointer is assigned a list cell. Furthermore, the value field of the list cell cannot be used until it is initialized. Within the main routine, the producer and consumer execute in parallel.

The modify-privileges interface is essentially the same as the declarative interface. The only difference is that the `value` field of the list cell is *not* `sync`. The modify-privileges interface protocol requires that shared values can only be modified by the procedure with modification privileges. Modify privileges are passed from the producer to the consumer when the `sync next`

```
// The value of the list element and the pointer to the
// next list cell are both sync variables

struct list {
  sync T value;
  struct list * sync next;
}
```

Figure 1: The list structure used to pass data between a producer and a consumer.

pointer is initialized. Thus we must ensure that the value component of the list is initialized before the next pointer is set.

Our final example is a message passing interface. In a message passing interface, we must have an entity to send a message to. Therefore, we will define the producer and consumer as user defined types. We associate a set of functions with each user defined type. Thus the `produce` function can be applied to a variable of type `producer`, while the `insert_queue` and `consume` operations can be applied to a variable of type `consumer`. The consumer also has a `get_queue` operation which is only accessible to variables of type `consumer`.

The main program creates a `producer` and `consumer` variable and applies the `produce` and `consume` operations to the `producer` and `consumer` respectively. The `producer` inserts a data value directly into the queue of the `consumer` by calling applying the `insert_queue` operation. The `consumer` then extract the data values and processes them. The operations on the queue must be made atomic to prohibit `insert_queue` and `get_queue` operations from occurring simultaneously.

References

[Ack82] William B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, feb 1982.

[Agh86] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AT80] Arvind and R.E. Thomas. I-Structures: An efficient data structure for functional languages. Technical Report TM-178, MIT, 1980.

[CK92] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. Technical Report Caltech-CS-TR-92-13, California Institute of Technology, 1992.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[CT91] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Bartlett and Jones, 1991.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[FC92] Ian Foster and K. Mani Chandy. Fortran M: Modular Fortran for parallel programming. Technical report, Argonne National Laboratory, 1992.

[FO90] Ian Foster and Ross Overbeek. Experiences with bilingual parallel programming. In *The Proceedings of the Fifth Distributed Memory Computer Conference*, 1990.

[FO91] Ian Foster and Ross Overbeek. Bilingual parallel programming. In *Proceedings of the Third Workshop on Parallel Computing and Compilers.* MIT Press, feb 1991.

[FT90] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[Gre87] Steve Gregory. *Parallel Logic Programming in PARLOG.* International Series in Logic Programming. Addison-Wesley, 1987.

[Lam91] Leslie Lamport. Temporal logic of actions. Technical report, DEC-SRC, 1991.

[Mar85] Alain J. Martin. The Probe: An addition to communication primitives. *Information Processing Letters*, 20:125–130, April 1985.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(1):319–340, 1976.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Sei91] Charles Seitz. *Developments in Concurrency and Communication*, chapter 5, pages 131–200. Addison Wesley, 1991.

[Sha86] Ehud Shapiro. Concurrent Prolog: A program report. *IEEE Computer*, 19(8):44–58, August 1986.

[TA90] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *ACM Operating Systems Review*, 24(3), July 1990.

[Tak89] Akikazu Takeuchi. How to solve it in Concurrent Prolog. Unpublished note., 1989.

[TE68] L. Tesler and H. Enea. A language for concurrent processes. In *Proceedings of AFIPS SJCC*, number ANL-91/38, 1968.

[Ued86] Kazunori Ueda. Guarded horn clauses. In *Logic Programming '85*, pages 168–179. Springer-Verlag, 1986.

```
producer(list * sync * ptr) {
  // A producer iterates allocating a new list cell, storing the pointer to
  // it into the sync next pointer from the previous iteration and
  // initializing the value field of the list cell.
  list * tmp;      // tmp is a pointer to a list cell
  while (1) {
    tmp = new list;                    // Allocate a new list cell
    tmp->value = producer_value();     // Initialize the value being produced
    (*ptr)->next = tmp;                // Pass modify privileges
    ptr = & (tmp->next);       // Get a pointer to the next field
  }
}

consumer(list * sync ptr) {
  // Iterate over the list created by the consumer.  Because they
  // are both sync, we have to wait for both the value and the
  // next pointer to be initialized before continuing.
  while (1) {
    consume_value(ptr-> value);
    ptr = ptr->next;
  }
}

main() {
  list * sync X;
  // Run the producer and consumer in parallel.  The consumer waits for
  // the list pointer X to be assigned a value.  The producer is passed
  // a non-sync pointer to the list so that it dosn't have to wait.

  par { producer(& X); consumer(X); }
}
```

Figure 2: A producer/consumer example using a declarative interface.

```
// Produce a value by sending it directly to the consumer
struct producer {
  produce(consumer * ptr) {
      while (1) { ptr->insert_queue(producer_value()); }
  }
}

// A consumer is a user defined data type with three operations associated with
// it.
struct consumer {
  atomic insert_queue(T); // Insert a value into the consumers queue
  void consume(list sync * ptr) // Consume the values put in the queue
     {
       while (1) { consume_value(get_queue()); }
     }
private:
   atomic T get_queue();   // Extract a value from the queue.
}

main() {
  producer P;                        // Create a producer object
  consumer C; // Create a consumer object

   // Start the producer and consumer
  par { P.produce( &C ); C.consume(); }
}
```

Figure 3: A producer/consumer example using a message passing interface.